UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería informática

TRABAJO FIN DE GRADO

DevOps para automatización de Gitlab en alta disponibilidad

Javier Dompablo Tobar
Tutor: Miguel Ángel Mora Rincón

Julio 2018

DevOps para automatización de Gitlab en alta disponibilidad

AUTOR: Javier Dompablo Tobar
TUTOR: Miguel Ángel Mora Rincón

Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2018

Resumen

La administración y gestión de los sistemas de computación, disruptivamente, permiten capacidades nunca vistas en el sector. El Cloud Computing, permite un uso ágil y eficiente de los recursos en red, con una mayor capacidad de computación. Este factor, unido a los procesos de automatización que permiten los despliegues de dichos recursos de manera paralela y automatizada, han convertido estas tecnologías en el punto clave de las TIC.

En los últimos tiempos, las tecnologías han evolucionado a un modelo de código abierto que las hacen más accesibles. El desarrollo de plataformas completamente distribuidas a nivel de software, únicamente implican costes derivados de los recursos físicos. Aprovechado las capacidades que el código abierto nos ofrece, el objetivo de este proyecto es el desarrollo de una plataforma de control de versiones de código, y además permita al usuario usar modelos de entrega continua.

Las buenas prácticas definen que la implementación de estas tecnologías se administre de manera automatizada, tanto para la parte del despliegue de la infraestructura, como para los elementos que la conforman dentro de un marco de Cloud Computing.

En el desarrollo nos apoyaremos en distintas tecnologías basadas en la virtualización, despliegue automatizado, herramientas de control de versiones, tecnología de contenedores y bases de datos distribuidas.

Finalmente, se realizarán unas mínimas pruebas para comprobar el buen funcionamiento de la plataforma, y estudiar el comportamiento del servicio mediante un ejemplo de construcción de código.

Palabras clave

Alta disponibilidad, Cloud Computing, automatización, virtualización, control de versiones, contenedores, devops, integración continua, despliegue continuo.

Abstract

The administration and management of computer systems, disruptively, allow capacities never seen in the sector. Cloud Computing enable a quick and efficient use of network resources, with a major computing capacity. This factor, connected with the automation processes that allows the deployments of these resources concurretly and automatically, have turned these technologies into the key point of ICT.

Recently, technologies have evolved to an open source model that makes them more accessible. The development of completely distributed platforms at the software level only implies costs produced by physical resource. Taken advantage of the capabilities that open source offers us, the goal of this project is the development of a code version control platform, and also allows the user to use continuous delivery models.

Best practices defined that the implementation of these technologies are managed automatically, both for the part of the infrastructure deployment, and for the elements that formed it within a Cloud Computing framework.

In the development we will rely on different technologies based on virtualization, automated deployment, version control tools, container technology and distributed databases.

Finally, a minimum of tests will be carried out to verify the proper functioning of the platform, and to study the performance of the service by an example of code construction.

Keywords

High Availability, Cloud Computing, automation, virtualization, versión control, containers, devops, continuous integration, continuous delivery

Agradecimientos

A Miguel Ángel por sus conocimientos

A Conrado por apoyarme

A Sara por su comprensión

A mi familia por saber esperar

A la EPS por hacerme madurar

INDICE DE CONTENIDOS

INDICE DE FIGURAS

1 Introducción

1.1 Motivación

Desde que comencé la carrera de Ingeniería Informática en la Escuela Politécnica Superior (EPS) de la Universidad Autónoma de Madrid (UAM), he tenido interés en los sistemas de la información. Gracias a los 4 años de becario en los laboratorios de la EPS, junto con los técnicos, he tenido acceso a los CPDs de la escuela. Siempre me ha llamado la atención la administración de estos sistemas y el diseño de arquitecturas que tienen como objetivo ofrecer un alto rendimiento y una alta disponibilidad.

En el mundo laboral, no se tiene la suerte de poder visitar estos sitios, ya que la mayoría no se encuentran en España y el acceso está más que restringido. Aun así, el Cloud Computing y la administración de sistemas se ha convertido más en un hobby que en una profesión.

Una de las cosas que siempre he echado en falta en la universidad, ha sido el uso de herramientas de control de versiones en las prácticas. Realizar entregables dentro de un marco de control de versiones, ayudaría tanto a los alumnos; a la hora de gestionar el código de manera colaborativa, como también al profesorado; para tracear aquellos posibles errores que puedan surgir en el desarrollo de los entregables. De la misma manera, el profesor podría comprobar de una manera muy sencilla, el nivel de colaboración de un alumno en las prácticas.

Actualmente, estas herramientas no solo son usadas por desarrolladores, sino que perfiles más orientados a gestión, recurren a ellas para comprobar el estado del proyecto. Por ello, aplicar estas técnicas a el mundo universitario, favorece las capacidades de un alumno a su salida al mundo laboral.

Recientemente, estas herramientas han sabido adaptarse a un mundo que desarrolla aplicaciones de forma ágil. Muchas de estas herramientas, llevan integrado lo que muchos entienden como una filosofía de trabajo, que permite abstraerse de ciertas tareas repetitivas que no focalizan la verdadera labor de un desarrollador. Estas tareas como la construcción, empaquetado, despliegue y pruebas; son perfectamente automatizables.

Quizás estas técnicas automáticas queden fuera del contexto universitario cuando el objetivo de las prácticas es reforzar este tipo de conocimientos o simplemente no sean aplicables a ciertos leguajes de programación. No obstante, ayudaría a facilitar el proceso de corrección de las prácticas por parte de los docentes y ampliaría el conocimiento del desarrollo del código.

1.2 Objetivos

El objetivo de este proyecto surgió con el fin de facilitar un servicio a la comunidad universitaria, más concretamente a los estudiantes de ingeniería de la Escuela Politécnica Superior (EPS) de la Universidad Autónoma de Madrid (UAM) de una herramienta de control de versiones para las prácticas en los laboratorios. La mayoría de las soluciones actuales son a través de pagos de licencias por persona y los costes derivados de este tipo de plataformas en ocasiones son inasumibles. Este proyecto, se ofrece como una solución de código libre que pretende sufragar los elevados costes, desarrollando la plataforma.

Como requisito principal para una plataforma que pretende dar servicio a una gran cantidad de usuarios y accesibilidad continuada, los requisitos son la alta disponibilidad y alto rendimiento. Por ello deberemos investigar cómo se encuentra el panorama actual con respecto a estas dos características.

Con esta visión clara, en primer lugar es analizar que infraestructuras son las más adecuadas para este tipo de implementaciones que se ofrecen en el mercado. Evaluaremos la posibilidad de usar clouds públicas y sus ventajas.

Otro requisito, por el volumen de la plataforma, es automatizar todo el proceso, desde la construcción hasta las configuraciones. Este paso, tocará tomar conocimiento de las herramientas se ajustan más al desarrollo de este proceso.

Por último, investigar que tecnologías de satisfacen el problema de distribución de datos, distribución de las cargas de tráfico, la disponibilidad de la aplicación.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- Capítulo 2: Estado del arte: pretende dar una visión global del paradigma de desarrollo actual en las TIC acotando diversos temas como los microservicios, los contenedores, la automatización y el Cloud computing.
- Capítulo 3: Diseño: análisis de los requisitos del proyecto y como se han resuelto mediante una visión de subsistemas.
- Capítulo 4: Desarrollo: la implementación de los requisitos de la fase de diseño. En esta sección se incluyen algunos ejemplos de código.
- Capítulo 5: Integración, pruebas y resultados: las pruebas de la fase de desarrollo. Estas se dividen en pruebas unitarias y pruebas de integración.
- Capítulo 6: Conclusiones y trabajos futuros: visión general del proyecto tras su desarrollo y posibles mejoras.

2 Estado del arte

2.1 Sistema de alta disponibilidad

El concepto de alta disponibilidad configura una serie de técnicas de diseño de un sistema computacional, donde su característica principal es ofrecer una continuidad temporal. Esto permite que los usuarios que hagan uso del sistema, tengan la posibilidad de crear, modificar, eliminar recursos del sistema de manera continua y sin perder el acceso a dicha información. La alta disponibilidad viene sujeta a todos los elementos tanto físicos como no físicos. En ocasiones, se puede ver comprometida por elementos dentro del sistema que como resultado de una mala gestión, pueden implicar inactividad en el servicio.

2.1.1 Evolución de los sistemas de alta disponibilidad

La transformación digital, ha renovado a grandes empresas que desarrollaban servicios que eran utilizados por los usuarios en su ordenador personal a servicios online. Esto implica, que muchas de las empresas han tenido que renovar sus infraestructuras y formas de trabajar para adaptarse a estos cambios. Estos servicios, son consumidos a través de internet por miles de usuarios concurrentes y diseñar una plataforma de alto rendimiento y de alta disponibilidad es algo habitual en el sector.

Un ejemplo de adaptabilidad a observar, es el caso de Netflix[1]. El flujo de datos que Netflix ha experimentado tras su salida a nivel mundial, requiere de plataformas de alto rendimiento. Esto solo se consigue reduciendo las latencias entre el emisor y el receptor, con una arquitectura escalable que permita asumir la gran demanda en momentos concretos.

Las aplicaciones que ofrecen servicios web, necesitan apoyarse en otras aplicaciones que les garanticen protocolos de tolerancia a fallos. Algunos de los servicios más conocidos en el panorama actual son HAProxy, NGINX o Keepalived.

En este punto, podríamos clasificar la alta disponibilidad en dos grandes grupos:

- Alta disponibilidad en infraestructura: los recursos físicos o virtuales son replicados a consecuencia de fallos en la infraestructura.
- Alta disponibilidad en aplicaciones: el fallo en aplicaciones donde la disponibilidad se realiza asignando responsabilidades a otros procesos activos dentro del sistema mientras este se recupera.

Para que un sistema preste servicio continuo debe de cumplir un mínimo de características como la redundancia en infraestructura. Los nodos de computación dentro de un sistema no deben ser únicos y tienen que estar replicados a un mínimo de razón de dos. Más adelante veremos que existen aplicativos que necesitan a razón de tres por sus características de comunicación y algoritmos de decisión. No únicamente los nodos de computación deben de replicarse, sino que también los elementos que los rodean.

[1] https://media.netflix.com/es/company-blog/completing-the-netflix-cloud-migration

2.1.2 Cloud Computing

Las grandes empresas, acuden a tecnologías de Cloud Computing donde abstraerse de los costes derivados de la compra de hardware para acudir a un modelo orientado al pago por uso. En este punto, empresas como Amazon, Google y Microsoft han sabido cubrir las necesidades empresariales creando *Clouds* públicas accesibles a todo el mundo.

La característica principal de estas plataformas, es que ofrecen fiabilidad del servicio contante en el tiempo y nos aseguran una disponibilidad del 99,9999% del tiempo útil. Decir que esta situación en algunos casos es ideal, debido a que no se cuentan tiempos de mantenimiento ni de circunstancias extraordinarias. La principal ventaja, es facilitar la gestión de los recursos físicos a un coste extraordinario. El diseño, la escalabilidad y el mantenimiento de la infraestructura a nivel de servicio, permite que la evolución de los proyectos tecnológicos tome un ritmo mayor, esto, en conjunto con técnicas de automatización, permite que el proceso sea más ágil y eficiente que nunca.

El Cloud Computing ha facilitado en gran medida la disponibilidad de los datos en distintas localizaciones geográficas, ofreciendo latencias más bajas. El coste computacional se ha reducido considerablemente gracias al pago por uso de los elementos virtuales, esto favorece la creación de plataformas más escalables tanto horizontalmente con verticalmente.

Servicios simples como el email o una máquina virtual son algunos ejemplos de estos servicios. Estas pueden tener un carácter público, privado o mixto, y ser usadas en cualquier parte del mundo.

El modelo clásico de Cloud Computing lo conforman 3 capas [1]:

- SaaS (*Software as a Service*)
- PaaS (*Platform as a Service*)
- Iaas (*Infrastructure as a Service*)

Dada la infinidad de servicios en red que actualmente están operando, nos centramos en la capa de IaaS que aplica en este proyecto.

La tendencia a usar virtualización de sistemas de computación en la nube es cada vez más popular. Muchos proveedores púbicos ofrecen ofertas cada vez más competentes y permiten al usuario autoabastecerse de la infraestructura de una manera sencilla y rápida. Algunos de los proveedores más importantes en el panorama actual son AWS (*Amazon Web Services*), Google Cloud Platforms o Microsoft Azure.

Algunos de los servicios más básicos que ofrecen son:

- Computación virtual
- Redes virtuales
- Almacenamiento en red
- Balanceo de carga

2.2 Microservicios

Se define como microservicio al paradigma de desarrollo de aplicaciones que da lugar a un software atómico y de desarrollo pequeño. El principio fundamental es desglosar aquellas arquitecturas complejas, en arquitecturas más sencillas.

La historia reciente del desarrollo de aplicaciones implica cambios en las arquitecturas [2]:

- **Arquitecturas monolíticas**: todo el código está mezclado y los cambios en cualquier parte de ellas pueden llevar a un fallo en todo el sistema. Es complejo para los equipos de desarrollo que trabajan sobre el mismo código.
- **Arquitecturas SOA**: pretenden conectar los distintos componentes entre ellos. Pero el paradigma de programación se basa en los mismos principios debido a la necesidad de comunicarse con el resto de los equipos para coordinar los cambios.
- **Arquitecturas de microservicios**: ofrece un desarrollo y coordinación entre los equipos, es más ágil al tratarse de aplicaciones más pequeñas. Como contrapartida, estas arquitecturas implican no solo el desarrollo del servicio sino de la API.

Los desarrollos dentro del sistema se encuentran desacoplados, dando flexibilidad a los desarrolladores a implementar distintos lenguajes de programación. Podemos encontrar algunos microservicios desarrollados en NodeJS, en Java o en Python. Esta flexibilidad también se da en los recursos externos que requieren las aplicaciones. Cada módulo independiente dentro del sistema puede contar con su propia base de datos en MySQL, PostgesSQL o Redis o también su propio sistema de servicio web como Nginx o Apache. De esta manera no sobrecarga el sistema en conjunto[2].

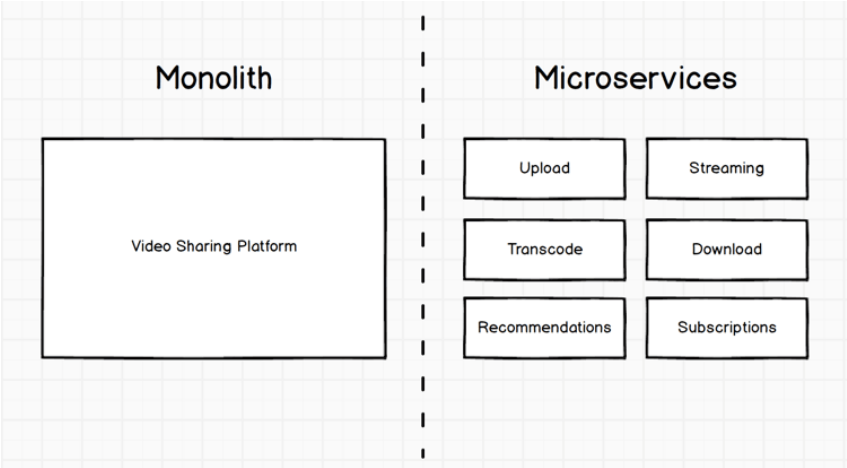

Ilustración 1. Comparación entre arquitecturas monolíticas y microservicios.

En cuento a la comunicación entre microservicios, existe una corriente purista que establece que los módulos no se comuniquen entre sí. Simplemente se espera de ello un *output* con un determinado *input*. En la realidad, este esquema se queda obsoleto a la hora de hablar de funcionalidad, ya que si se conserva los protocolos de comunicación adecuados, puede llegar a ser enriquecedor para los distintos módulos. El modelo típico de comunicación distribuida

[2] https://www2.deloitte.com/es/es/pages/technology/articles/microservicios.html

entre microservicios es HTTP/REST, estos reciben peticiones a través de una interfaz pública mediante una petición tipo REST, que es convertida a una respuesta JSON, entendible por el lenguaje de programación en el que se desarrolla el servicio. Es responsabilidad del programador integrar los DTOs correspondientes al lenguaje de programación para interpretar la información de entrada.

La gestión de estas comunicaciones debe ser coordinada por un elemento externo que atienda las peticiones y las distribuya a lo largo del sistema. En este punto, existen dos visiones para este propósito:

- Orquestación: este componente se encarga de la coordinación de las llamadas entre los microservicios. Quizás la más popular y usada en el ámbito empresarial. El mantenimiento de estos componentes es relativamente sencillo.
- Coreografía: en este caso, se utilizan elementos que permiten la comunicación a través de eventos. Estos mensajes se dejan en un componente común que se encarga de coordinar cuando una tarea ha finalizado para dar paso a la siguiente.

Al tratarse de elementos atómicos con tendencia al crecimiento, existen herramientas comunes que permiten una gestión eficaz en estas arquitecturas:

- Registry o Discovery Service (Registro): el objetivo principal es registrar todas aquellas nuevas instancias que se incorporan al sistema y de esta manera tener un conocimiento de la localización física o lógica del componente. En el caso de que algún otro componente necesite atacar a otro componente, utilizará esta herramienta para dirigir su petición.
- Configuration Service (Servicio de configuración): almacena la configuración de todos los microservicios. Este debe de ser único para todo el sistema.
- Load Balancer (Balanceador de carga): es el encargado de distribuir equitativamente la carga en los servicios. Este puede estar incluido dentro del registro como único componente.
- Circuit Breaker (Sistema de tolerancia a fallos): forma de programar métodos de *failover*. En caso de fallo de algún elemento del sistema, este componente corta la comunicación entre los componentes afectados para que sea otro el encargado de la comunicación con el elemento no afectado.
- Edge Service (Servicio perimetral): centraliza todas las peticiones en un único punto a modo de *proxy* inverso. Determina cuál de los microservicios se está intentando apuntar a través de la clase de petición que se realiza a este componente redirigiéndolo hacia donde corresponda.
- Logs centralizados: cada microservicio genera unos *logs* que están centralizados en un mismo lugar para facilitar la trazabilidad en caso de error.

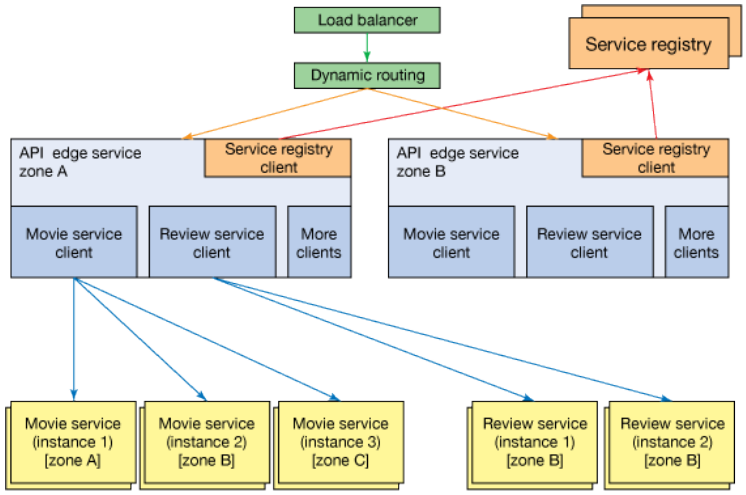

Ilustración 2. Ejemplo de arquitectura basada en microservicios.

Los microservicios ofrecen grandes ventajas con respecto a las arquitecturas monolíticas:

- Diseño tolerante a fallos.
- Desarrollos independientes que no afectan al conjunto del sistema.
- Escalabilidad independiente.
- Reusabilidad de los componentes.
- Visión de producto y no de proyecto.
- Automatización de los procesos de entrega (test, construcción, despliegue)

2.3 Contenedores

Actualmente, el incremento en el desarrollo de microservicios ha contribuido a que muchas empresas tengan que evolucionar para dar soporte tecnológico a este nuevo paradigma. Empresas como Google, Redhat o Canonical a través de las tecnologías de virtualización ligera o contenedores pretenden cubrir estas necesidades.

El concepto de *container* no es algo nuevo, ya en 1970 con la salida de Unix V7 se introdujo el sistema de *chroot* que pretendía cambiar los directorios de los procesos hijos en el sistema de ficheros para isolizarlos. Más adelante FreeBSD introdujo los *"jails",* que no era más que una separación de los servicios en un entorno compartido. No se hizo una primera referencia a la palabra *container* hasta el año 2004 con Solaris Containers. Ya en 2006, Google lanzo una herramienta para la gestión de los recursos llamada "Proccess Containers", que posteriormente se llamaría Control cGroups que pretendía controlar los recursos de las máquinas donde se alojaban ciertos procesos. Pero fue una tecnología basada en Linux Kernel la que lanzó LXC estandarizando las herramientas, plantillas y bibliotecas necesarias para interactuar con estas dos tecnologías. Usado muy habitualmente a día de hoy cumple con los principios básicos de la arquitectura de contenedores que muchos proveedores han reinventado [3].

Se define como *container* (contenedor) a aquellos procesos isolizados dentro de un sistema operativo que a través de una imagen previamente generada, contiene todas aquellas dependencias que una aplicación precisa. Al tratarse de un único fichero autocontenido, este es portátil y puede ser replicado en tantos entornos como se desee.

Los principios básicos de la arquitectura de contenedores son [4]:

- **cGroups**: los grupos de control permiten de manera nativa en los sistemas Linux la asignación de recursos a un proceso o a un conjunto de procesos. Esto junto con systemd que en este caso nos permite inicializar el espacio de nombres del usuario y la gestión del proceso.
- **Namespaces**: los espacios de nombre permiten asignar identificadores únicos a ciertos usuarios o grupos de usuarios. En el contexto de los contenedores, esto permite asignar ciertos privilegios a ciertos usuarios o grupos únicamente dentro del contenedor.

Docker se ha convertido en uno de las herramientas de gestión de contenedores más popular. Técnicamente es un *daemon* (*docker-engine*) que corre en la maquina *host* y que a través de un protocolo cliente-servidor es manipulado mediante REST API. Mediante CLI podemos realizar acciones como crear, modificar, destruir y monitorizar contendores. Generalmente, el cliente y el servidor se encuentran en la misma máquina. Las imágenes la podemos generar manualmente a través de un fichero de configuración llamado Dockerfile, pero siempre dependerán de una base generada previamente. Esta imagen, la podemos obtener desde el servicio público de imágenes Docker[3] o desde un *registry* privado. El *daemon* es el encargado de generar el contenedor que podemos ver en la maquina como un proceso más.

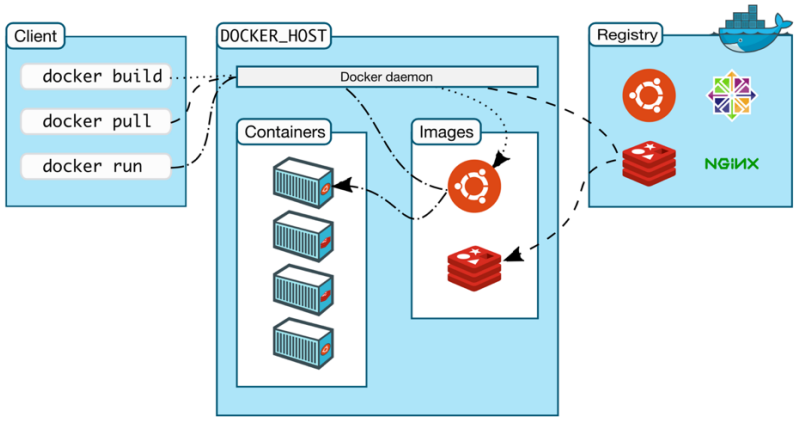

Ilustración 3. Ejemplo de proceso de generación de un contenedor.

El lenguaje de programación que usa para implementar el código del daemon es GO. Como comentábamos anteriormente, docker utiliza las tecnologías de namespaces (espacios de nombres) para crear el entorno de ejecución del docker. Estos son algunos de los valores que usa docker para la isolizacion del container [4].

- The pid namespace: Process isolation (PID: Process ID).
- The net namespace: Managing network interfaces (NET: Networking).
- The ipc namespace: Managing access to IPC resources (IPC: InterProcess Communication).
- The mnt namespace: Managing filesystem mount points (MNT: Mount).
- The uts namespace: Isolating kernel and version identifiers. (UTS: Unix Timesharing System).

[3] www.dockerhub.com

[4] https://docs.docker.com/engine/docker-overview/#the-underlying-technology

Al mismo tiempo el uso de control *groups* (cGroups) permiten limitar los recursos que Docker usa en cada contenedor. Algún ejemplo es limitar la memoria disponible para un contenedor en concreto.

Las imágenes de Docker están compuestas por una consecución de capas sobrepuestas unas con otras que partiendo de una imagen base, permita incluir nueva funcionalidad a la imagen. Esto lo provee el sistema de ficheros Union File System. La ventaja de esta tecnología, es que con archivos que provienen de sistemas de ficheros distintos, se pueda montar una unidad lógica que se vea como un único sistema de fichero.

En el caso de Docker, incluye un sistema de orquestación a nivel de container que facilita la gestión de los procesos activos, incluyendo características de replicación que garantizan un número de copias en el clúster contante. Muchos proyectos ya están utilizando este modelo desde hace años para mantener alta disponibilidad del servicio. Decir que en muchos casos como microservicios de api rest desarrollados en Python o en Nodejs es un buen caso de uso pero en algunos casos como la replicación de datos persistentes con postgreSQL dado el alto grado de inconsistencias que se produciría en la integridad del dato.

2.4 Entrega continua

La filosofía de entrega continua o CD (*Continuous delivey*) [3], viene muy arraigada al desarrollo de microservicios. Este enfoque de ingeniería del software, tiene como objetivo reducir los tiempos de generación de versiones de los componentes mediante la automatización de los procesos de construcción, empaquetado, despliegue y pruebas.

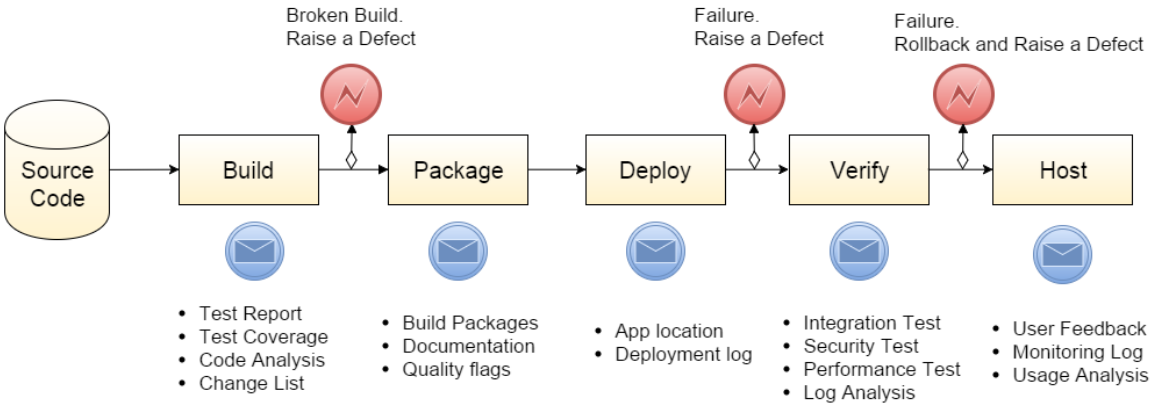

Ilustración 4. Proceso de entrega continua.

3 Diseño

3.1 Introducción

La idea de este proyecto surgió con el fin de facilitar un servicio a la comunidad universitaria, más concretamente a los estudiantes de ingeniería de la Escuela Politécnica Superior (EPS) de la Universidad Autónoma de Madrid (UAM) de una herramienta de control de versiones para las prácticas en los laboratorios. La mayoría de las soluciones actuales son a través de pagos de licencias por persona y los costes derivados de este tipo de plataformas en ocasiones son inasumibles. Este proyecto, se ofrece como una solución de código libre que pretende sufragar los elevados costes, desarrollando la plataforma.

El estudio comenzó analizando los requisitos del sistema. La alta disponibilidad y la automatización para la reutilización de la plataforma, han sido pilares fundamentales en el desarrollo. En consecuencia, ha sido necesario investigar que tecnologías del panorama actual satisfacían estas dos necesidades. Plantear una arquitectura estable en el tiempo, requiere un esfuerzo de diseño previo a la implementación y que facilita en gran medida el trabajo posterior.

3.2 Análisis de requisitos

Referenciamos todos aquellos requisitos del sistema.

3.2.1 Requisitos funcionales

RF-1 Acceso a la interfaz de Gitlab por parte del usuario.
RF-2 Crear, modificar y eliminar un repositorio en Gitlab.
RF-3 Generar *commits* en un repositorio en Gitlab.
RF-4 Lanzar un trabajo en integración continua.
RF-5 Lanzar un trabajo en despliegue continuo.

3.2.2 Requisitos no funcionales

RNF-1 Reservar IP estática pública para acceso a través de Internet.
RNF-2 Reservar dominio público para acceso a través de Internet.
RNF-3 Acceso a través de una única IP.
RNF-4 Alta disponibilidad en acceso al sistema (Balanceo de carga).
RNF-5 Alta disponibilidad en los datos del sistema (Redundancia de datos).
RNF-6 Elasticidad en volumen de datos.
RNF-7 Alta disponibilidad en sesiones de usuario (Réplica de cachés de usuarios).
RNF-8 Soporte a contenedores.

3.3 Propuesta tecnológica

Para dar solución a este tipo de plataformas que plantean varios problemas de alta disponibilidad, en primer lugar proponemos dividir el sistema en diferentes subsistemas y abordar cada uno de ellos de manera individual. Más adelante, el concepto de subsistema tomará forma de clúster de alta disponibilidad o conjunto de computadoras con un fin específico.

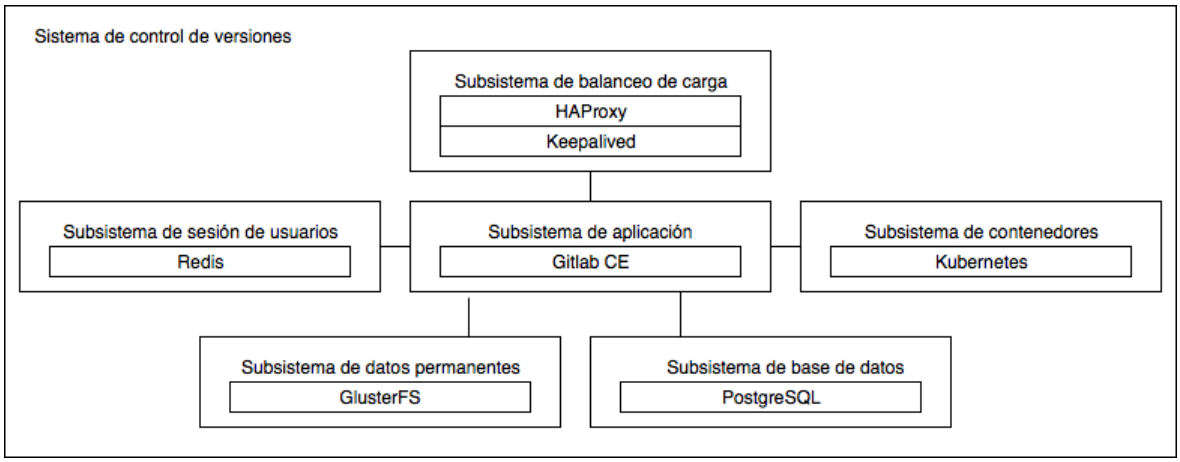

Ilustración 5. Diagrama inicial del sistema dividido en subsistemas.

3.3.1 Subsistema de balanceo de carga

Este subsistema de balanceo de carga tiene como objetivo ofrecer acceso público y distribuir las peticiones de usuarios al sistema. La VIP (Virtual IP) debe ser accesible a través de Internet de manera pública *[RNF-1]* y estar asociada a un servicio DNS que resolviera el nombre de dominio *[RNF-2]*. En caso de varios nodos de acceso, nos apoyamos en protocolos de coordinación (*HeartBeat*) que mantiene una VIP (*Virtual IP*) que trabajando en modo maestro-esclavo, mantiene siempre activo uno de ellos.

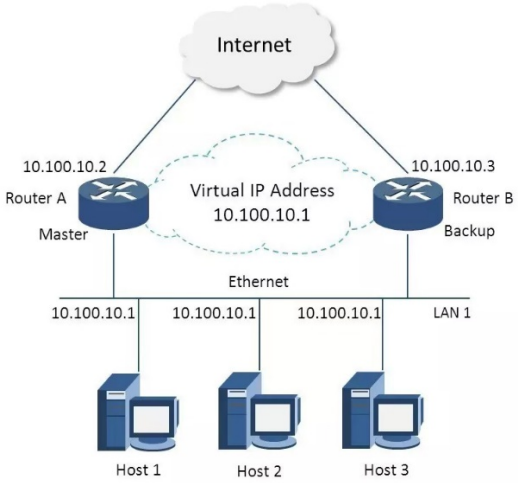

Ilustración 6. Protocolo dinámico de asignación de VIP.

Las peticiones de usuario a través de protocolos TCP a un puerto especifico, redirigen hacia las réplicas de los nodos de aplicación. Estos procesos se conocen como procesos de *failover* y garantizan que el tráfico siempre es enrutado a cualquiera de los nodos activos disponibles.

HAProxy

HAProxy es un software de distribución libre, que tiene como objetivo el balanceo de carga de tráfico. Además nos ofrece un sistema de *proxying* a través del protocolo TCP para aplicaciones basadas en HTTP.

Esta tecnología monitoriza a través de *Health Checks* (chequeos de estado de puertos TCP) a los servidores que pretenden servir. Las peticiones entran en un instante del tiempo en cualquier de los nodos, y bajo cualquier circunstancia de fallo en uno de los servidores, el balanceador redirige el tráfico.

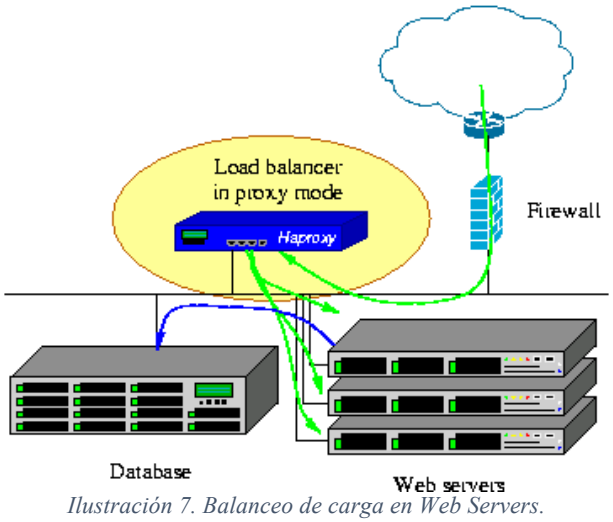

Ilustración 7. Balanceo de carga en Web Servers.

En primer lugar, se define en configuración, la capa de *Frontend* que se ofrece como *proxy* HTTP. En este caso se debería elegir una tecnología de *Heartbeat* entre las dos máquinas de *Haproxy*, para asignar una *VIP* (*Virtual IP*) como único punto de entrada. Esta capa generalmente está establecida en el puerto 80 cuando el protocolo es HTTP o 443 cuando el protocolo es seguro por HTTPS, donde puede ser definida en cualquier puerto TCP. La capa de *Backend,* por configuración define los servidores a balancear. En este caso, es habitual utilizar el *hostname* de la máquina y un puerto TCP, dependiendo del puerto en el que escuche la aplicación.

Existen distintos tipos de balanceo de carga en función de la capa de protocolo en la que nos encontremos.

- Balanceo de capa de transporte (*Layer 4*): es uno de los más habituales. El balanceo se realiza a través de IP y puerto. Elegida en el proyecto por simplicidad.
- Balanceo de capa de aplicación (*Layer 7*): más complejo. El balanceo se realiza en base al contenido de la petición de usuario.

La selección del nodo candidato, dependerá del algoritmo previamente definido en configuración. Es posible elegir cualquiera de estos 3 algoritmos:

- *Round Robin:* es el algoritmo por defecto sino se especifica otro por configuración. Determina el nodo siguiente mediante una lista de elementos de manera secuencial. Cuando acaba de recorrer la lista, comienza de nuevo.
- *Leastconn*: determina que nodo tiene menor número de conexiones. Esto ayuda a mantener sesiones más largas.
- *Source*: determina el servidor en base basándose en el hash de la IP origen de usuario. Esto ayuda a que los usuarios mantengan la sesión en el mismo servidor.

3.3.2 Subsistema de aplicación

El subsistema de aplicación pretende ser parte *frontend* del sistema. En este caso particular, nuestro *frontend* es una aplicación de control de versiones, aunque el desarrollo de la plataforma es agnóstico a la aplicación. Es objetivo de este proyecto, dar una solución plausible a otros tipos de aplicaciones desarrollas en otros lenguajes de programación y con fines diferentes, que requieran alta disponibilidad.

Gitlab

Gitlab es un software libre en su versión Gitlab CE (*Comunity Edition*) y empresarial en su versión Gitlab EE (*Enterprise Edition*) de control de versiones de código y desarrollo colaborativo para grupos de trabajo basado en Git. Está escrito en Ruby, por dos ucranianos *Dmitriy Zaporozhets y Valery Sizov,* que fundaron la empresa Gitlab Inc en octubre de 2011.

Cuando se habla de Gtilab, es necesario comprender el uso que hace de la tecnología Git. Esta tecnología permite el control absoluto de los cambios realizados en un código fuente dentro de un marco de trabajo. Generalmente, este protocolo es implementado por aplicaciones de terceros (Gitlab, GitHub, etc) que ofrecen una interfaz para acceder al software. La generación de cambios en el código fuente se realiza localmente para posteriormente actualizarlos en un servidor centralizado encargado de evaluar las modificaciones con respecto a la última versión.

Algunos de los conceptos básicos de Git son:

- Ramas (*Branches*): generan un entorno de trabajo duplicado del código. Cuando se realiza un *fork* del código actual, puedes realizar cambios en los ficheros de esa rama.
- *Master:* rama por defecto, donde se realizan todas las actualizaciones de código.
- *Commits:* cuando se genera un cambio, es necesario tomar una instantánea para actualizar los cambios al servidor remoto.
- *Pull Request:* es la confirmación de los cambios realizados en el código. Un punto de encuentro donde analizar los cambios efectuados y verificar que es posible actualizar la rama *Master.*

La suite de Gitlab dispone de herramientas que ayudan al desarrollador software a generar un *pipeline* completo de CI (*Continuous Integration*) y CD (*Continouos Delivery*) que agilizan los desarrollos. Se define *pipelines* como una consecución de fases que, partiendo de un código fuente, se llega a una versión *release* de una aplicación. Alguna de las fases intermedias para llegar a esta fase final pueden ser la construcción, empaquetado, test de aplicación y despliegue en un entorno de pruebas que definen el ciclo de vida software. Los *pipelines* son lanzados a través de la generación de *commits* en el desarrollo habitual del software.

Para que las fases del *pipeline* se realicen de manera ágil, Gitlab coordina estas tareas a través de *executors*[5]. El concepto que usa Gitlab para materializar esta técnica es el uso de *runners.* El portfolio de posibles *executors* o tecnologías donde construir estas tareas, van desde Shell, SSH, VirtualBox; hasta tecnologías de contenedores como Docker, Docker Machine y Kubernetes. En este proyecto, se ha elegido Kubernetes por su posibilidad de replicación y concurrencia derivada de la orquestación de los contenedores.

[5] https://docs.gitlab.com/runner/executors/

3.3.3 Subsistema de datos de sesión de usuario

El objetivo del subsistema de sesiones de usuarios es replicar aquella información que la plataforma almacena en caché y replicarla a lo largo de los nodos. En caso de fallo y rebalanceo a los nodos de aplicación, esta información estará accesible y consumible desde cualquier nodo.

Redis

Redis es un software libre de almacén de datos en memoria. Esta característica, ha colocado a Redis en una de los sistemas de almacenamiento de datos más populares del panorama. A diferencia de las bases de datos tradicionales como MySQL, PostgreSQL o RDBMS que los datos se almacenan en tablas, en Redis la información se almacena de forma no estructurada [6]. Los tipos de datos más habituales son las siguientes:

- Cadenas de caracteres (*String)*
- Tablas *Hash (Hash)*
- Listas *(Lists)*
- Conjuntos *(Sets)*
- Conjuntos Ordenados *(Sorted sets)*

Incorpora mucha funcionalidad aparte del almacenamiento en memoria:

Base de datos clave-valor *NoSQL*

Las claves pueden llegar a ser tanto secuencias binarias, como archivos de imagen. Los tamaños de las claves son variables aunque el tamaño máximo es 512MB. Esta limitación, se debe al bajo rendimiento que puede tener el sistema en base a tener claves tan grandes en memoria o realizar búsquedas comparando una secuencia de *bytes* tan larga. Por el contrario, utilizar claves cortas no favorece la legibilidad y por tanto las búsquedas por claves no son tan naturales.

Caché de datos

Algunos de los usos más habituales son:

- **Caché de páginas web**: almacenar los ficheros estáticos de las páginas webs dentro de una caché acelera el acceso a dicha información. De esta manera, se reduce el acceso a los recursos y por tanto reducción de la carga.
- **Almacenamiento de sesiones de usuarios**: las sesiones se almacenan en alta disponibilidad. En muchas ocasiones esto se traduce a que el usuario no es consciente de parada de servicio por fallo en el sistema.
- **Caché de bases de datos**: las *queries* más recurrentes se almacenan en caché para reducir tiempos de respuesta.

Redis PUB/SUB

Redis PUB/SUB en un bróker de mensajería basado en colas de mensajes. Permite mediante la suscripción a un canal de comunicación, la extracción de eventos que genera un publicador Estos patrones de mensajería pretenden realizar una validación de los datos de entrada, transformación de los mensajes para que sean legibles por otra aplicación y el posterior ruteo de los datos de salida. Este tipo de aplicaciones favorece el desacoplamiento de las aplicaciones, no siendo necesario que estén programadas en el mismo paradigma de programación.

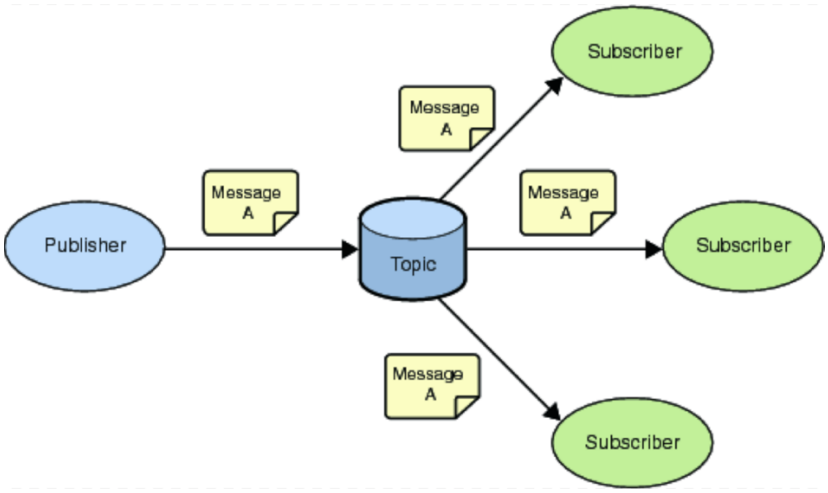

Ilustración 8. Broker de mensajería basado en publicador/suscriptor.

Los comandos más básicos son:

- Message: recibe un evento de tipo mensaje
- Subcribe: suscripción a un canal especifico.
- Unsubcribe: cancelación de la suscripción de un canal especifico.

Existe un mecanismo de expiración de claves que determina el tiempo que las claves se almacenan en memoria.

Redis admite persistencia de datos en disco de los datos que se encuentran en memoria. Esto nos permite, utilizar el sistema como una base de datos *NoSQL* tradicional con datos recientemente usados rápidamente accesibles en memoria y datos no tan recientemente usados en disco. Predeterminadamente, podemos usar varios algoritmos de persistencia para dar solución a distintos problemas:

- RDB (*snapshooting*): guarda en disco los registros en memoria en un intervalo de tiempo predefinido. A mayor tiempo, mayor la perdida de datos en caso de fallo. A tiempos muy bajos, los recursos pueden afectar al rendimiento del sistema.
- AOF (*appending*): guarda en disco los registros en un fichero. Con este algoritmo te permite recuperar todos los registros.

Es posible deshabilitar la persistencia de datos. De esta manera, el uso del software es equivalente a una cache. Esto implica que en el caso de fallo y reinicio de la maquina donde se almacena la información, esta se perdería.

3.3.4 Subsistema de datos permanentes

El subsistema de datos permanente tiene como objetivo replicar los datos estáticos de los nodos de aplicación de manera distribuida a través de red. La accesibilidad en caso de fallo a los datos persiste cuando se monta un punto de montaje único. Asegurar la consistencia de los datos cuando son varios servicios los que realizan tareas de lectura y escritura, requiere de un sistema de archivos distribuidos conectado en red.

GlusterFS

GlusterFS es un sistema de archivos multi-escalable a través de protocolo TCP. La estructura básica de estos nodos es en formato cliente-servidor. Los archivos son accesibles a través del demonio que exponen los nodos de archivos y consumidos a través de los clientes en los nodos de aplicación. Lógicamente, lo que ve el nodo de aplicación en un punto de montaje con permisos de lectura/escritura. Como vemos en la figura, los ficheros se alocan en diferentes puntos de montajes del clúster de manera independiente, pero existe la posibilidad de replicar los datos en espejo para ofrecer alta disponibilidad.

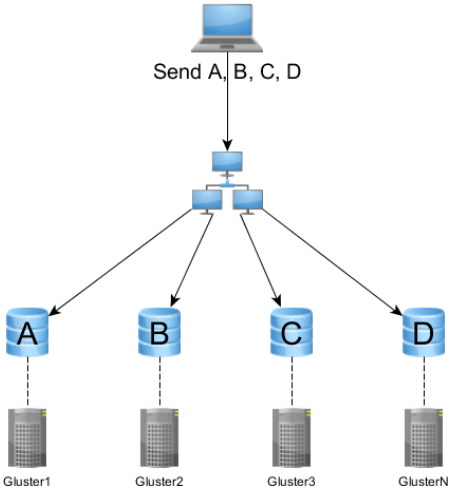

Ilustración 9. Distribución de datos con Gluster.

3.3.5 Subsistema de bases de datos

El subsistema de bases de datos tiene como objetivo la persistencia de los datos relacionados con la administración de usuarios del subsistema de aplicación.

PostgreSQL

PostgreSQL es un sistema de bases de datos basados en datos relacionales. Este sistema admite replicación de datos a lo largo de un sistema distribuido en un modelo maestro-esclavo. Esto pretende aumentar el rendimiento y la redundancia de datos.
Por sí solo, PostgeSQL no cuenta con la tecnología suficiente para la replicación de datos, apoyándose en otras tecnologías para garantizar la alta disponibilidad. En concreto se basa en dos tecnologías:

- Repmgr (*Replication Manager*): es una herramienta de código abierto que gestiona el *failover* y la replicación dentro de los clúster de PostgreSQL.
- Consul: servicio de descubrimiento de nodos en el clúster. Esto permite alertar al resto de nodos fallos ocurridos dentro del clúster.

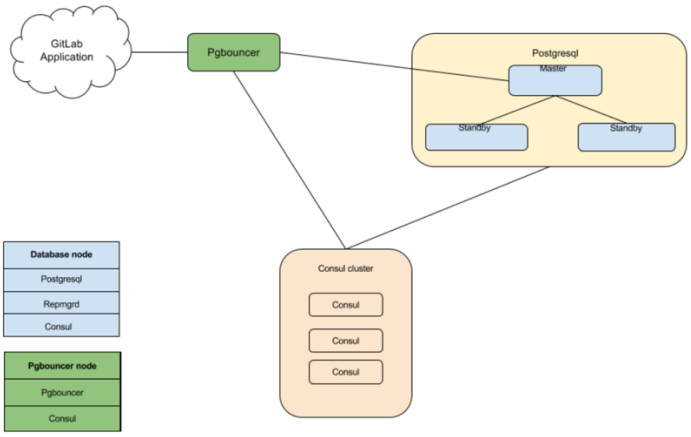

Ilustración 10. Alta disponibilidad en bases de datos distribuidas.

3.3.6 Subsistema de contenedores

El subsistema de contenedores no es algo necesario para ofrecer garantías en alta disponibilidad. El caso de uso del subsistema de aplicación, trae consigo una integración con kubernetes para ofrecer servicios de entrega continua.

Kubernetes

Kubernetes (K8S) es considerado un orquestador de contenedores que facilita el escalado, el despliegue y la administración de contenedores. Algunos de aspectos a destacar de esta tecnología son :

- **Despliegues automáticos**: en el caso en que se realicen modificaciones en cualquiera de los contenedores, K8S realiza los despliegues de manera progresiva controlando en todo momento si los cambios producidos en el contenedor producen fallos.
- **Descubrimiento de servicios y balanceo de carga:** dispone de servicios externos de descubrimiento de contenedores mediante la asignación de IP fijas y un servicio DNS. Además, al monitorizar los recursos de cada contenedor, puede balancear la carga.
- **Autoescalado:** la monitorización de los contendores contribuyen a especificar que valores son susceptibles a asignar nuevos recursos mediante la creación de nuevos contenedores. De esta manera, la plataforma puede escalar tanto para aumentar las capacidades como para reducirlas.
- **Gestión de configuración y secretos:** la configuración de la aplicación se puede actualizar sin necesidad de reconstruir el contenedor y las passwords se almacena oculta.

3.3.7 Automatización con Ansible

Ansible [6] es una plataforma de código abierto que nos permite orquestar los procesos de configuración y administración de las máquinas virtuales de manera automatizada. Mediante conexiones SSH, es capaz de configurar múltiples nodos en paralelo sin ninguna configuración adicional en los nodos remotos. La principal ventaja, es su integración con muchos proveedores de infraestructura. Permite una gran versatilidad, por ejemplo a la hora de generar inventarios con información de las máquinas o reutilizar módulos con tareas previamente definidas. Su uso es simple, ya que combina el formato de serializarían *YAML* y ficheros dinámicos en formato *Jinja2*. Esta conjunción de ficheros, no es más que una lista de tareas de configuración llamadas *playbooks* en un grupo determinado de máquinas.

En este proyecto, se ha decidido automatizar los despliegues de la plataforma con esta tecnología, por distintos motivos:

- Simpleza en la escritura de las tareas en las máquinas remotas.
- Rendimiento multi-nodo.
- Módulos de despliegue reutilizables.
- Módulos de generación de infraestructura.
- Integración con *Vagrant* y *Virtualbox*.

3.4 Arquitectura

Hasta ahora hemos podido ver que partes conforman nuestra plataforma desde un punto de vista de subsistemas. La integración de los múltiples subsistemas en una arquitectura única, implica analizar la disposición de las tecnologías previamente descritas.

En primer lugar, hacemos una aproximación desglosando los subsistemas en recursos replicados. No se tiene en cuenta la reutilización de recursos, sino que se disponen como una forma lógica de cómo interactúan entre los nodos de la plataforma [7].

Analizando la figura que presentamos a continuación:

- 2 nodos para balanceo de carga
- 3 nodos para aplicación
- 3 nodos de datos
- 3 nodos de contenedores

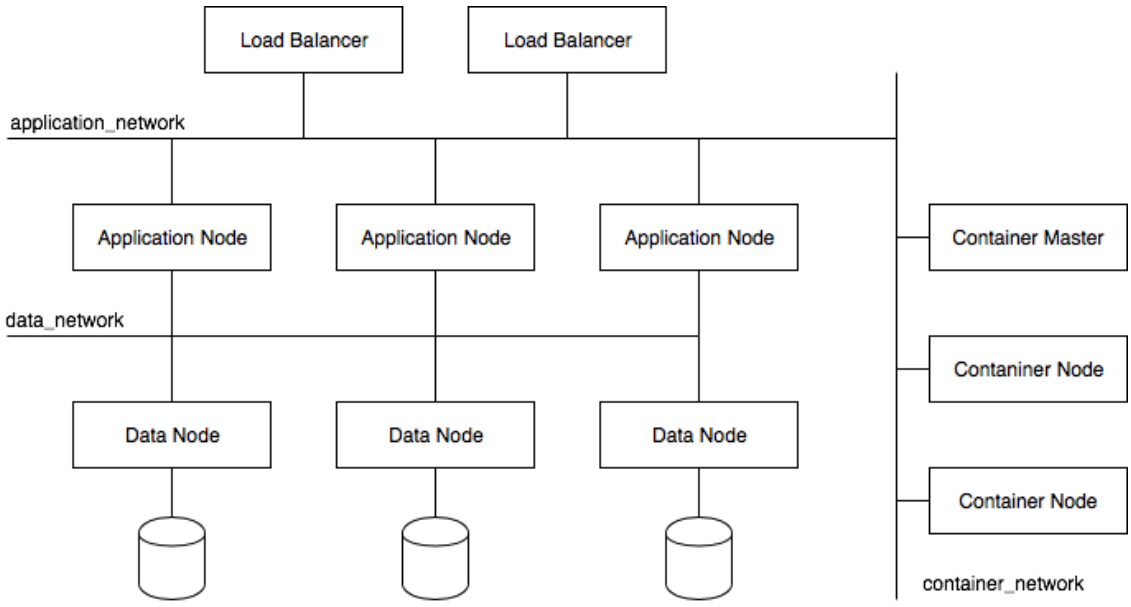

Ilustración 11. Distribución de subsistemas en un modelo normalizado.

Una vez descrito un modelo normalizado de la arquitectura, nos disponemos a analizar los requisitos estructurales de las tecnologías que hemos propuesto para resolver el problema.

Analizamos las capas de la arquitectura una a una:

- **Capa de balanceo de carga:** HAProxy replica los nodos a razón de 2 en modo activo-pasivo. De esta manera, en caso de caída de un nodo activo, se activaría el protocolo de *failover* mediante Keepalived, asignando la VIP al nuevo nodo activo.
- **Capa de aplicación y sesiones de usuario:** en la capa de aplicación se encuentra la aplicación de Gitlab. Según especificación, únicamente es necesario replicarla a razón de 2. En este caso, se ha tomado la decisión de compartir el clúster de sesiones de usuario en los mismos nodos. Esta decisión, pretende que los datos de caché sean rápidamente accesibles en el mismo nodo. El protocolo de *failover* de Redis, implica un mínimo de 3 nodos para la elección del nodo master[6].
- **Capa de datos persistentes y bases de datos:** las referencias de clústers en alta disponibilidad, sugieren que los datos no se encuentren dentro de los nodos donde van a ser consumidos. Además, los datos deben de separarse en un volumen físico del habitual del sistema operativo. Gluster, también trabaja en modo maestro-esclavo y requiere de 3 nodos para operar en alta disponibilidad. El mismo caso para PostgreSQL.
- **Capa de contenedores:** la capa de contendores no se presenta como un clúster de alta disponibilidad. La especificación, sugiere que los nodos Kubernetes master se repliquen a razón de 3 y los Kubenetes nodes tengan un mínimo de 3 nodos. Veremos más adelante, que el sistema ya soporta replicación de procesos a nivel de software y que por tanto no necesario desplegar más nodos.

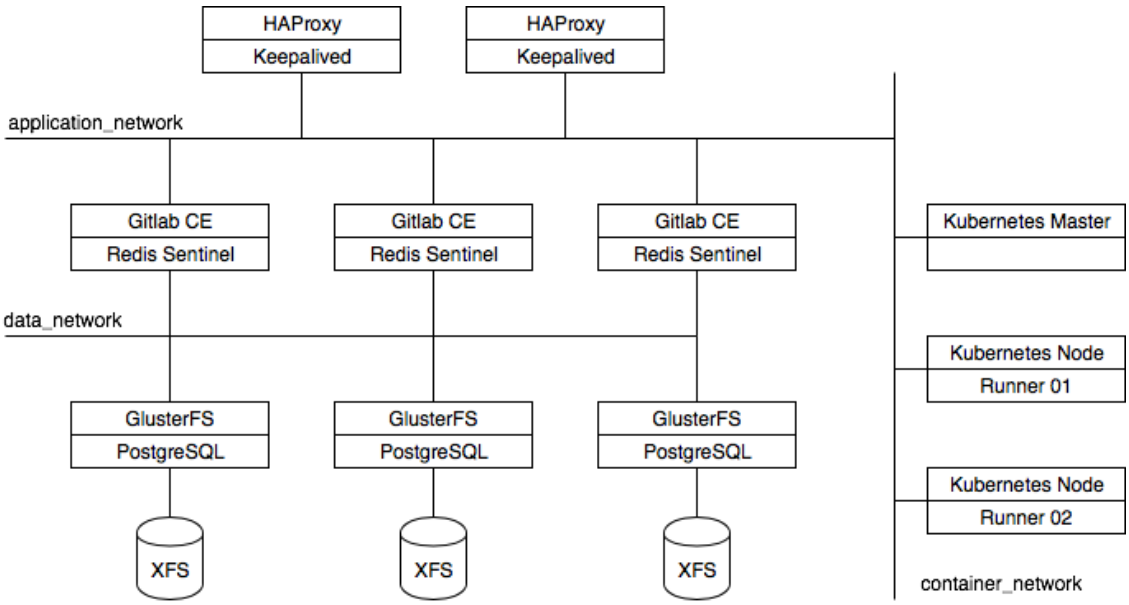

Ilustración 12. Diagrama definitivo basado en alta disponibilidad.

[6] https://en.wikipedia.org/wiki/Split-brain_(computing)

4 Desarrollo

4.1 Introducción

La fase desarrollo se divide en tres iteraciones en las que nos marcamos los objetivos de probar iterativamente los requisitos no funcionales del sistema. Como veremos, en cada etapa se incrementan las capacidades del sistema que parten de una fase anterior, en la que se intentan probar varios proveedores de infraestructura, una iteración de automatización, hasta la etapa de integración de los distintos clústeres para llegar a la solución final.

4.2 Desarrollo de la solución

Una de las partes más complejas del proyecto fue decidir sobre que infraestructura montar el proyecto. Como hemos comprobado en el estado del arte, existen múltiples *Clouds* que ofrecen servicios similares pero que a la larga pueden resultar costosas. En consecuencia, dejamos a decisión del futuro administrador de la plataforma, la opción de adaptar los scripts de Ansible para aprovecharse de las capacidades de otras *Clouds*.

4.2.1 Primera iteración

La primera iteración consistía en realizar pruebas de concepto con diferentes sistemas de infraestructuras. Para comenzar, acudimos a la herramienta Vagrant. Esta herramienta nos permite mediante un fichero de configuración, interactuar con el software de virtualización VirtualBox para generar las máquinas virtuales. La generación de la infraestructura es relativamente sencilla. Un fichero *Vagrantfile* con la especificación de las de las imágenes que utilizaríamos, así como la limitación de los recursos de cada máquina.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
domain = 'GitlabHA'

Vagrant.configure("2") do |config|
config.vm.define "{{server.name}}" do |nodeconfig|
    nodeconfig.vm.box = "{{server.os}}"
    nodeconfig.vm.box_url =  "https://vagrantcloud.com/" + "{{server.os}}"
    nodeconfig.vm.hostname = "{{server.name}}"
    nodeconfig.vm.network
:private_network,       :ip       =>       '{{server.ip}}',       :name       =>
'{{network.private_network}}', :adapter => 2, :auto_config => true

    nodeconfig.vm.provider :virtualbox do |vb|
      vb.customize  ["modifyvm",  :id,  "--cpuexecutioncap",  "50",  "--memory",
"{{server.memory}}"]
    end
end
```

Esta primera prueba de concepto ya incluía un nodo de aplicación (bundle de la Gitlab CE en *standlone*) y el clúster de contenedores conformado por un nodo de Kubernetes master y dos Kubernetes nodes.

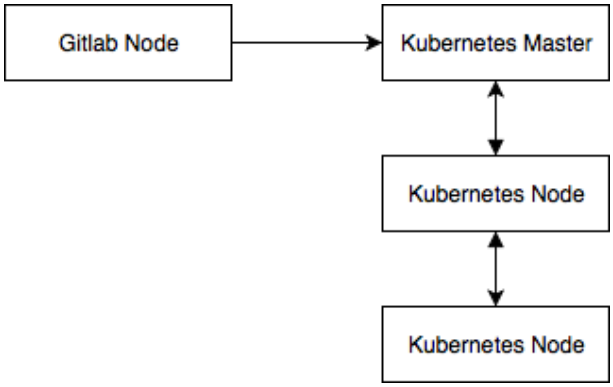

Ilustración 13. Arquitectura en POC local.

Decidimos realizar una prueba de concepto en un entorno de Cloud Computing pública. Google Cloud nos proporciona en este caso, un paquete de bienvenida para probar la plataforma con un límite de cuota que podemos consumir durante un año. Además se ha pensado en este proveedor porque de forma nativa ofrece servicios de Kubernetes, que facilitarían la implantación y el desarrollo de este proyecto. Para este caso, creamos un clúster de Kubernetes mediante la consola que nos proporciona Google. Rápidamente la prueba gratuita se consumía los recursos que teníamos disponibles y se tuvo que abandonar esta prueba.

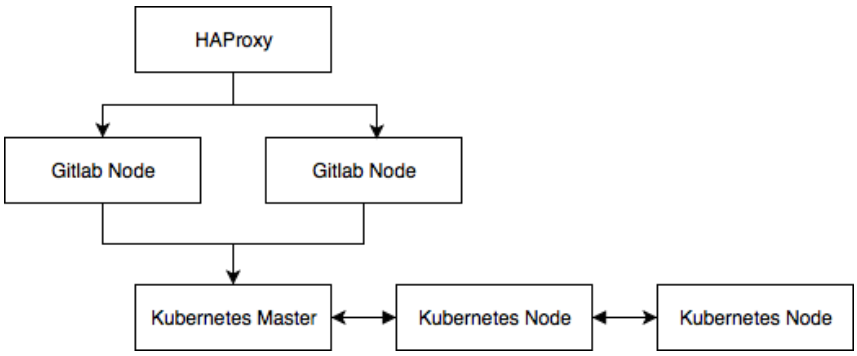

Ilustración 14. Arquitectura en POC d Google Cloud

En un tercera aproximación, ya la definitiva, consideramos la utilización de un rack con un número mayor de recursos que nos facilitaría el despliegue la plataforma. En este caso, como proveedor de infraestructura elegimos Virtualbox ya que nos permitía reutilizar los scripts de las pruebas de concepto que se hicieron en local y nos darían una estabilidad y funcionalidad que necesitábamos para la plataforma. Previamente en esta misma máquina, comprobamos las virtudes de LXD como infraestructura. Es este punto nos encontramos muchos inconvenientes, sobretodo en el escalado de privilegios y en la reutilización de los módulos del kernel en las capas superiores de LXC. Decir que los cliente de VirtualBox son más pesados y que en muchos de los casos no optimizan ni reutilizan los recursos que les ofrece el host pero que a la larga, contar con máquinas virtuales autocontenidas donde no dependes del host anfitrión para realizar ciertas tareas, beneficia la salud del clúster y simplifica los trabajos de mantenimiento de la misma.

En la fase de diseño se contempló el despliegue de dos balanceadores de cargar que mediante un protocolo de *HeartBeat* controlase el estado de la VIP. En plataformas Cloud Computing esto es posible contratando servicios como *elastic* IP en AWS o reservando una IP externa en Google Cloud. En este caso, VitualBox no nos ofrecía una solución clara ante este problema. En este punto se tomó la decisión de desplegar una solo máquina para facilitar el punto de entrada.

Problemas

- **Recursos de una máquina limitados**: El servicio de Gitlab genera una cantidad de procesos derivados de los componentes de bases de datos, datos en memoria, persistencia de datos, monitorización de procesos, etc. En cuanto al clúster de Kubernetes, contienen muchos elementos "dockerizados" como elementos de coordinación, acceso y coordinación de claves-valor que precisan un mínimo de recursos.
- **Recursos con costes elevados**: los recursos que estábamos generando en la plataforma de Google Cloud eran excesivamente costosos, lo que implicaba que en cualquier momento podríamos perder el derecho al acceso a la infraestructura.
- **Falta de automatización en el proceso de despliegue**: Vagrant nos provee de infraestructura de manera automática. Pero en este caso solo quedaría automatizada la parte de la infraestructura. Esta situación, no permite replicar la misma configuración en distintas máquinas virtuales o redesplegar en caso de fallo de alguna de ellas.
- **No cumple los principios de la alta disponibilidad**: la falta de recursos impedía escalar en infraestructura. En este punto perdíamos la tolerancia a fallos en el clúster así como una distribución de la carga en ciertos puntos. La redundancia tanto de los datos como de los elementos que mantienen la sesión de usuario se perdían por que no se encontraban en alta disponibilidad.

Conclusión

Existen múltiples soluciones que se adaptan a las necesidades de cada proyecto. En nuestro caso, alojar el contenido de la aplicación en una cloud pública no ha sido viable. Los datos quedarían a disposición de la seguridad que aplique la cloud pública en cada caso.

Al no automatizar el proceso, perdíamos la posibilidad de volver a replicar la infraestructura en cualquier otro entorno. La falta de automatización condenaba el despliegue, que en caso de fallo, deberíamos redesplegar manualmente. Por tanto, no podrimos asegurar dejar la plataforma en el mismo estado.

4.2.2 Segunda iteración

El objetivo de la segunda iteración del desarrollo fue la automatización de la plataforma. Como se ha indicado previamente, la herramienta que elegida para este fin ha sido Ansible. Como primera aproximación, definimos los scripts de despliegue de la infraestructura. Dado que contábamos con un fichero Vagrantfile, decidimos reaprovechar el contenido para darle flexibilidad, parametrizando mediante un fichero de configuración YAML, la declaración de las máquinas. En el anexo A podemos ver el fichero Vagrant completo.

```
nodes =
[
  {% for server in servers %}
    {   :hostname => '{{server.name}}',
        :ip => '{{server.ip}}',
        :box => '{{server.os}}',
        :memory => {{server.memory}}
     },
  {% endfor %}
]
```

```
servers:
  - name: kube-master
    ip: "172.16.0.2"
    memory: 2048
    cpu: 80
    os: "ubuntu/xenial64"
  - name: kube-minion01
    ip: "172.16.0.3"
    memory: 1024
    cpu: 80
    os: "ubuntu/xenial64"
  - name: kube-minion02
    ip: "172.16.0.4"
    memory: 1024
    cpu: 80
    os: "ubuntu/xenial64"
```

Ansible a través de su módulo de vagrant facilita la conectividad con las maquinas remotas. En este caso es vagrant el encargado de generar un inventario estático en función de la parametrización de las maquinas. El establecimiento de la conexión SSH, se realiza en función a la IP y puerto previamente acordado, así como el correspondiente intercambio de claves para establecer una conexión segura. Todo esto se almacena en el proyecto oculto que genera vagrant. Es este caso, es necesario especificar en el Vagrantfile que el fichero de la *provision* es mediante ansible.

```
  config.vm.provision "ansible" do |ansible|
    ansible.verbose = "v"
    ansible.playbook = "playbook-init.yml"
    ansible.extra_vars = {
      ansible_python_interpreter: "/usr/bin/python3",
    }
```

Por motivos de simplicidad, se generó manualmente una red interna de VirtualBox que conectaba todos los nodos de la plataforma. Aclarar que en este contexto de pruebas, la red suele ser uno de los puntos débiles del desarrollo. Para un entorno productivo, sería necesario organizar las redes en *VLANs* que discriminara el acceso a determinados recursos dentro de la plataforma. A nivel arquitectónico, se ha planteado una organización de red, pero a nivel práctico no se llegó a implementar.

Los servicios de la plataforma también son susceptibles a automatizar. Dentro de un proyecto Ansible, podemos especificar roles. Esta unidad atómica actúa como un conjunto de tasks que desempeñan unas tareas concretas. En ella se incluyen tanto ficheros estáticos, ficheros dinámicos, variables locales como la declaración de las tareas. Un ejemplo de la disposición de un role podría ser el siguiente:

```
ansible-role-glusterfs
|_defaults
  |_main.yml
|_meta
|_tasks
  |_configure.yml
  |_main.yml
  |_setup-Debian.yml
  |_setup-RedHat.yml
|_vars
```

La buenas prácticas nos indican que es conveniente que los roles sean una unidad reutilizable. Esto implica que los roles deben definir tareas separadas de instalación para plataformas Debian y RedHat. La configuraciones son elementos comunes en cómo se establece un servicio.

Los roles son referenciados por un fichero de declaración común que establece las llamadas a los roles en *hosts* específicos. Los *hosts* han sido previamente declarados en el inventario estático que genera vagrant por defecto en su integración con Ansible. En el anexo D se puede ver la declaración completa.

Ansible permite configurar ciertos aspectos del despliegue en un fichero de configuración llamado ansible.cfg. Algunos de esos parámetros son los siguientes.

```
[defaults]
hostfile = .vagrant/provisioners/ansible/inventory/vagrant_ansible_inventory
host_key_checking = False
gathering = smart
fact_caching_timeout = 86400
fact_caching = jsonfile
fact_caching_connection = ansible_fact_caching.cache
retry_files_enabled = False
deprecation_warnings=False

[ssh_connection]
scp_if_ssh = True
pipelining = True
```

El esquema en esta aproximación es algo diferente. En este caso la parte de los datos debía estar en alta disponibilidad y por tanto era necesario tanto generar la infraestructura como automatizar el despliegue de estos componentes. Nos informamos sobre cuáles eran las recomendación en la documentación una plataforma de Gitlab en alta disponibilidad y la nos proponía 5 tecnologías distintas.

Problemas

- **Algunos recursos no son automatizables:** como en el caso de los componentes de red de VirtualBox, algunos pasos son manuales debido a que no existe una integración.
- **No existen módulos para todos los casos de uso:** aunque el desarrollo va en aumento, en muchas ocasiones es necesario acudir a otros lenguajes declarativos para tareas más complejas.
- **La integración con VirtualBox depende de Vagrant:** hubiera sido un escenario más real el caso de realizar despliegues con un módulo de VirtualBox.
- Dependencia total de la versión de pyhton: nos encontramos problemas a la hora de poder desplegar Ansible por su dependencia a la versión Python 3 que no todas las distribuciones la incluyen.

Conclusión
El desarrollo de la automatización con Ansible ha sido rápido, intuitivo y nos ha facilitado tener una base operativa muy rápidamente. Gracias a la parametrización de los ficheros de los servicios, independizar y reutilizar los módulos nos ha permitido configurar cada máquina con los valores que le correspondían.

En un futuro, si el lector desea reutilizar el código, es posible gracias a la independencia de los ficheros de configuración de Ansible. Es posible la reutilización de los mismos roles que se usaron es esta plataforma pero en una infraestructura distinta[7].

4.2.3 Tercera iteración

La tercera iteración nos centramos en el proceso de despliegue y configuración de los servicios de la plataforma. Lo dividimos en los subsistemas acordados en el capítulo de diseño.

Subsistema de balanceo de carga
Este subsistema se vio afectado por las condiciones previamente comentadas. La necesidad de un punto único de entrada para simular una VIP. La configuración de HAProxy se llevó a cabo en único nodo que escuchaba peticiones en el puerto 80 y las redirigía a los subsistemas de aplicación que escuchaban en el mismo puerto.

```
frontend hafrontend
    bind *:80
    mode http
    default_backend habackend


backend habackend
    mode http
    balance roundrobin
    option forwardfor
    option httpchk HEAD / HTTP/1.1\r\nHost:localhost
    cookie SERVERID insert indirect
    server gitlab01 172.16.0.6:80 cookie gitlab01 check
    server gitlab02 172.16.0.7:80 cookie gitlab02 check
    server gitlab03 172.16.0.11:80 cookie gitlab03 check
```

Por un lado, la parte de Frontend escucha en todas las interfaces por el puerto 80 en protocolo HTTP. Sería necesario para futuros pasos configurar SSL para conexiones más seguras. Por otro lado, en el Backend declara los hosts disponibles. Decir que estas peticiones se realizan a través de IP y no de *hostname*, por tanto no es necesario tener declarado la referencia al hostname en */etc/hosts.* El algoritmo de selección de nodo es Round Robin. Se decidió el algoritmo por defecto ya que el mantenimiento de la sesión se iba a externalizar en otro servicio y no convenía utilizar cualquiera de los otros dos disponibles. El uso de cookie no nos ha ofrecido ninguna ventaja técnica. Aun así lo dejamos referenciado por que podría llegar a ser interesante su uso con el objetivo de prestar un mejor servicio y tener mayor visibilidad del usuario.

[7] https://docs.ansible.com/ansible/latest/modules/list_of_cloud_modules.html

Subsistema de aplicación
El subsistema de aplicación está conformado por los servicios que exponen la capa *frontend* de la plataforma. En concreto, Gitlab se apoya de todos estos

```
root@gitlab01:~# gitlab-ctl status
run: gitaly: (pid 24445) 894194s; run: log: (pid 17804) 905188s
run: gitlab-monitor: (pid 24456) 894194s; run: log: (pid 17826) 905188s
run: gitlab-workhorse: (pid 24469) 894193s; run: log: (pid 17832) 905188s
run: logrotate: (pid 11043) 1374s; run: log: (pid 17834) 905188s
run: nginx: (pid 24484) 894192s; run: log: (pid 17833) 905188s
run: node-exporter: (pid 24490) 894192s; run: log: (pid 17803) 905188s
run: postgres-exporter: (pid 24495) 894192s; run: log: (pid 17878) 905187s
run: postgresql: (pid 24575) 894191s; run: log: (pid 17768) 905189s
run: prometheus: (pid 24583) 894191s; run: log: (pid 17879) 905187s
run: sidekiq: (pid 24593) 894190s; run: log: (pid 17770) 905189s
run: unicorn: (pid 24601) 894190s; run: log: (pid 17769) 905189s
```

- Gitaly: nos ofrece una capa de gRPC de acceso a los repositorios.
- Gitlab-monitor: métricas de servicio Gitlab
- Gitlab-workhorse: manejador de peticiones de usuario.
- Logrotate: servicio de rotado de logs.
- Nginx: servidor web.
- Node-exporter: utilizado por phrometeus para exportar métricas de máquinas.
- PostgreSQL-exporter: utilizado por phrometeus para exportar métricas de bases de datos.
- PostgreSQL: base de datos de usuarios.
- Prometheus: Servicio de visualización de métricas.
- Sidekiq: servicio de trabajos en background
- Unicorn: servidor de aplicaciones.

Las peticiones web entran por el servicio NGINX que se encuentra escuchando en el puerto 80. Esta petición HTTP es redirigida al servicio de Gitlab-workhorse que discrimina entre peticiones del servidor de aplicación o peticiones de ficheros estáticos.

Subsistema de datos de sesión de usuario
El servicio de datos en memoria se ha desplegado en las maquinas de aplicación. Se encarga de consultar la cache local y determina los datos del usuario activo dentro de los 3 nodos. Sentinel, es el protocolo de *failover* que utiliza Redis para determinar donde replicar la información distribuida en memoria de las máquinas. Es requisito indispensable, que esta replicación se haga en 3 réplicas del clúster[8]. Esto ha determinado el número de nodos de la capa de aplicación, que con un mínimo de 2 nodos ya podría dar servicio en alta disponibilidad.

La distribución de redis de alta disponibilidad, se dispone en formato maestro-esclavo. Un nodo es elegido como maestro, mientras que el resto se mantiene a la espera en el caso de detección de un fallo. En nuestro caso, el quorum está fijado a 2 ya que contamos con 3 nodos. Este valor determina el mínimo número de nodos activos para que se continúe dando servicio. Las escrituras se realizan en el nodo Master. En caso de fallo del nodo master, el cliente podría continuar escribiendo en el nodo que master recientemente perdido en el tiempo en el que se reconfiguran las escrituras en memoria. Para mitigar este problema, es

[8] https://en.wikipedia.org/wiki/Split-brain_(computing)

necesario especificar una variable que indique la detección de las escrituras en el nodo master cuando no escriba en cualquiera de los nodos esclavos.

```
min-slaves-to-write 1
min-slaves-max-lag 10
```

La configuración de sentinel:

```
maxclients 4064
sentinel known-slave gitlabHA 172.16.0.11 6379
sentinel known-slave gitlabHA 172.16.0.7 6379
sentinel known-sentinel gitlabHA 127.0.0.1 26379
sentinel known-sentinel gitlabHA 172.16.0.11 26379
sentinel known-sentinel gitlabHA 172.16.0.7 26379
sentinel current-epoch 0
```

La configuración de los nodos de sentinel, permiten obtener información de los nodos dentro del clúster. Es tarea de nodo esclavo, monitorizar el estado contra el nodo master. En el caso en que no reciba respuesta del nodo Master en el intervalo de tiempo especificado, el nodo esclavo manda por broadcast un mensaje SDOWN (*Subjective down*). A continuación, si el nodo master se encuentra en estado inactivo, emite otro mensaje de ODOWN (*objetive DOWN*)

Subsistema de datos permanentes

El subsistema de datos permanente se encuentra localizado en instancias separadas de los nodos de aplicación. Por diseño, lo datos deben de estar replicados en una zona no compartida donde originalmente se generan los datos por seguridad.

El almacenamiento de datos requiere de un daemon de servidor basado en GlusteFS en las maquinas donde se almacenan los datos y de un daemon de cliente que se encargue del montaje de dicho volumen en local. El nodo de aplicación, atacará directamente al montaje único en dicho punto. En esta ocasión, ha sido relativamente sencilla la implementación debido a que existe un módulo de Ansible dedicado a este fin.

```
- name: Ensure Gluster brick and mount directories exist.
  file: path={{ item }} state=directory mode=0775
  with_items:
    - "{{ gluster_brick_dir }}"
    - "{{ gluster_mount_dir }}"

- name: Configure Gluster volume.
  gluster_volume:
    state: present
    name: "{{ gluster_brick_name }}"
    brick: "{{ gluster_brick_dir }}"
    replicas: 2
    cluster: "{{ gluster_cluster }}"
    host: "{{ inventory_hostname }}"
    force: yes
  run_once: true

- name: Ensure Gluster volume is mounted.
  mount:
    name: "{{ gluster_mount_dir }}"
    src: "{{ inventory_hostname }}:/{{ gluster_brick_name }}"
    fstype: glusterfs
    opts: "defaults,_netdev"
    state: mounted
```

En primer lugar es necesario indicar el path donde el server monta el volumen, en nuestro caso sobre */tmp/gitlab.* Para esta implementación solo hemos tenido que montar un único *brik* que lo localizamos en */var/lib/data.* Así como el nombre del *brick*

```
- name: Ensure Gluster volume is mounted.
  mount:
    name: "{{ gluster_mount_dir }}"
    src: "{{ gluster_master }}:/{{ gluster_brick_name }}"
    fstype: glusterfs
    opts: "defaults,_netdev"
    state: mounted
```

La implementación en la parte de cliente también fue sencilla. Únicamente necesitábamos indicarle el punto de montaje y la IP del nodo master del clúster así como el nombre del brick.

Como vemos tras la salida de una simple consulta de la información de un volumen es la siguiente:

```
gluster> volume info

Volume Name: data
Type: Replicate
Volume ID: 4e38f1c9-ac88-4198-99a3-6baa26b435d9
Status: Started
Snapshot Count: 0
Number of Bricks: 1 x 3 = 3
Transport-type: tcp
Bricks:
Brick1: 172.16.0.9:/var/lib/data
Brick2: 172.16.0.10:/var/lib/data
Brick3: 172.16.0.12:/var/lib/data
Options Reconfigured:
nfs.disable: on
performance.readdir-ahead: on
transport.address-family: inet
```

Subsistema de bases de datos

Este subsistema no se llegó a implementar en su plenitud por falta de tiempo. Los nodos de aplicación ya contaban con una bases de datos PostgreSQL propia, y por ello las pruebas su realizaron duplicando los usuarios en las 3 instancias.

Subsistema de contenedores

Este subsistema pretende dar soporte a los runners de Gitlab para generar tareas de construcción, empaquetado y despliegue dentro de un entorno de contenedores.

En primer lugar, fue necesario instalar la versión de Docker 17.03.2-ce por problemas de compatibilidad con versiones más recientes. El siguiente paso, fue desplegar la configuración del nodo Kubernetes Master. Es necesario realizar la instalación del nodo master ya que luego serán el resto de nodos los que se adjuntaran al clúster mediante un comando *join*. En una infraestructura de alta disponibilidad, sería necesario la creación de 3 nodos Master y un mínimo de 3 nodos tolerante a fallos. En este caso (por ahorro de recursos) solo se ha desplegado un nodo master y dos nodos. Veremos más adelante que con la replicación de contenedores, no es necesaria más infraestructura para dar alta disponibilidad.

```
root@kube-master:~# kubectl get nodes --all-namespaces
NAME           STATUS    ROLES     AGE       VERSION
kube-master    Ready     master    17d       v1.10.3
kube-minion01  Ready     <none>    17d       v1.10.3
kube-minion02  Ready     <none>    17d       v1.10.3
```

A continuación, vemos los servicios que ofrece el nodo master en Kubernetes

```
root@kube-master:~# kubectl get po --namespace kube-system
NAME                                 READY     STATUS    RESTARTS   AGE
etcd-kube-master                     1/1       Running   1          17d
kube-apiserver-kube-master           1/1       Running   1          17d
kube-controller-manager-kube-master  1/1       Running   1          17d
kube-dns-86f4d74b45-hpsd8            3/3       Running   3          17d
kube-flannel-ds-lr9nx                1/1       Running   1          17d
kube-flannel-ds-nxhql                1/1       Running   1          17d
kube-flannel-ds-wdbt5                1/1       Running   1          17d
kube-proxy-84jvr                     1/1       Running   1          17d
kube-proxy-v2hnd                     1/1       Running   1          17d
kube-proxy-vg7np                     1/1       Running   1          17d
kube-scheduler-kube-master           1/1       Running   1          17d
```

El servicio de overlay elegido para esta implementación ha sido flannel. Este servicio nos ofrece lo básico para conectar los nodos del clúster a través de una subred que se conecta a la interfaz de bridge que ofrece Docker. En el caso necesitar definir políticas de red y enrutar el tráfico dentro de la misma, se debería acudir a otras implementaciones.

El objetivo de este subsistema, es generar los runners en los cuales correrán las tareas que asigne Gitab. En primer lugar, es necesario la creación de un *namespace* para englobar todas las configuraciones. Lo llamamos *gitlab.* Es necesario definir un fichero de variables (*ConfigMap*) para desacoplar la parametrización con la construcción de la imagen y flexibilizar el comportamiento del container una vez desplegado.
La manera en la que los runners se autentican en el servicio de Gitlab es a través de *token*. La propia interfaz nos proporciona este *token*, que en nuestro caso lo podemos tratar como un secret dentro del contenedor encriptado en base64 para ser utilizado posteriormente. Llegados al punto de la implementación de la aplicación, tocaba tomar la decisión si los runners debían de mantener o no el estado. Esto significa si los *pods* deberían mantener datos de manera persistente. La documentación oficial nos proponía un despliegue tradicional sin replicación (aunque lo sugiere). Dado que tenemos que almacenar la variable de manera persistente en el nodo, se dio la necesidad de configurar los runners a través de la API de gestión de aplicaciones con estado, *StatefulSet.* Algunos parámetros más relevantes son:

```
réplicas: 2
containers:
  - image: gitlab/gitlab-runner:v10.4.0
    name: gitlab-ci-runner

    ports:
      - containerPort: 9100
        name: http-metrics
        protocol: TCP

root@kube-master:~# kubectl get po --namespace gitlab
NAME                 READY     STATUS    RESTARTS   AGE
gitlab-ci-runner-0   1/1       Running   0          9d
gitlab-ci-runner-1   1/1       Running   0          9d
```

5 Integración, pruebas y resultados

Las pruebas en esta plataforma van orientadas a comprobar los sistemas de alta disponibilidad de los clúster independiente a modo de pruebas unitarias y terminar con unas pruebas de integración de la plataforma completa a través de un caso de uso tipo. En este caso no se realizan pruebas de estrés y pruebas de seguridad.

5.1 Pruebas unitarias

5.1.1 Pruebas en HAProxy

Para comprobar el balanceo de carga de una aplicación HTTP, es necesario lanzar peticiones desde el punto de acceso para comprobar que la petición se realiza en un de ellos. Seguidamente tirar el servicio para que el balanceador realice el chequeo de caída del nodo. En ese caso el resultado que se espera es que la petición se realice en nodo activo.

Comprobamos que el servicio de balanceo tiene IP

```
vagrant@haproxy:~$ ip a
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
group default qlen 1000
    link/ether 08:00:27:6a:28:9e brd ff:ff:ff:ff:ff:ff
    inet 172.16.0.8/24 brd 172.16.0.255 scope global enp0s8
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe6a:289e/64 scope link
       valid_lft forever preferred_lft forever
```

La ip de esta máquina es la 172.16.0.8 que se encuentra en escucha en el puerto 80.

```
vagrant@haproxy:~$ netstat -toupan | grep 80
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      -
off (0.00/0/0)
```

Desde los nodos de aplicación, monitorizamos los logs de NGINX de Gtilab

```
root@gitlab02:~# tail -f /var/log/gitlab/nginx/gitlab_access.log
172.16.0.8 - - [21/Jun/2018:18:37:04 +0000] "GET / HTTP/1.1" 302 97 ""
"Wget/1.17.1 (linux-gnu)"

root@gitlab02:~# gitlab-ctl stop
ok: down: alertmanager: 0s, normally up
ok: down: gitaly: 1s, normally up
ok: down: gitlab-monitor: 0s, normally up
ok: down: gitlab-workhorse: 1s, normally up
ok: down: logrotate: 0s, normally up
ok: down: nginx: 0s, normally up
ok: down: node-exporter: 1s, normally up
ok: down: postgres-exporter: 0s, normally up
ok: down: postgresql: 1s, normally up
ok: down: prometheus: 1s, normally up
ok: down: sidekiq: 1s, normally up
ok: down: unicorn: 0s, normally up

root@gitlab01:~# tail -f /var/log/gitlab/nginx/gitlab_access.log
172.16.0.8 - - [21/Jun/2018:18:38:51 +0000] "GET / HTTP/1.1" 302 97 ""
"Wget/1.17.1 (linux-gnu)"
```

Como vemos la petición wget solicitada en local primero redirige la petición hacia el nodo 2 para posteriormente rebalancear al nodo 1 cuanto este no está activo.

5.1.1 Pruebas en Redis

En primer lugar, comprobamos si Redis está corriendo en los 3 nodos de aplicación. En este caso, de todas las funcionalidades de Redis, solo probaremos la que hemos implementado, guardar datos en memoria.

Realizamos un simple ping para comprobar que el servicio está escuchando peticiones.

```
root@gitlab01:~# redis-cli -h 172.16.0.6 -a gitlab ping
PONG
root@gitlab02:~# redis-cli -h 172.16.0.7 -a gitlab ping
PONG
root@gitlab03:~# redis-cli -h 172.16.0.11 -a gitlab ping
PONG
```

Para realizar una prueba simple, introducimos un valor en el nodo que se encuentra en master (en los esclavos no se puede escribir). Y comprobamos como se ha replicado en el resto de nodos esclavos

```
vagrant@gitlab03:~$ redis-cli -h 172.16.0.11 -a gitlab incr mycounter
(integer) 1
root@gitlab02:~# redis-cli -h 172.16.0.7 -a gitlab get mycounter
"1"
root@gitlab01:~# redis-cli -h 172.16.0.6 -a gitlab get mycounter
"1"
```

5.1.1 Pruebas en Gluster

Para realizar las pruebas sobre GlusterFS vamos a escribir un fichero simple en uno de los nodos y comprobar que se replica bien en el resto de nodos.
En primer lugar comprobamos la información del volumen de datos para determinar donde debemos de escribir nuestro fichero.

```
gluster> volume info

Volume Name: data
Type: Replicate
Volume ID: 4e38f1c9-ac88-4198-99a3-6baa26b435d9
Status: Started
Snapshot Count: 0
Number of Bricks: 1 x 3 = 3
Transport-type: tcp
Bricks:
Brick1: 172.16.0.9:/var/lib/data
Brick2: 172.16.0.10:/var/lib/data
Brick3: 172.16.0.12:/var/lib/data

Options Reconfigured:
nfs.disable: on
performance.readdir-ahead: on
transport.address-family: inet
```

Vamos a escribir la salida del nombre del hostname bajo /var/lib/data del nodo gluster01 y comprobaremos que re replica dicha información en el nodo glsuter02 y gluster03.

```
root@gluster01:~# hostname > /var/lib/data/hostname
root@gluster01:~# cat /var/lib/data/hostname
gluster01
root@gluster02:~# cat /var/lib/data/hostname
gluster01
root@gluster03:~# cat /var/lib/data/hostname
gluster01
```

5.1 Pruebas de integración

Para las pruebas de integración, vamos a crear un proyecto base en PHP y lanzar la ejecución de un runner para comprobar que hace unas mínimas pruebas.

El primer paso es generar un repositorio para poder introducir el código fuente

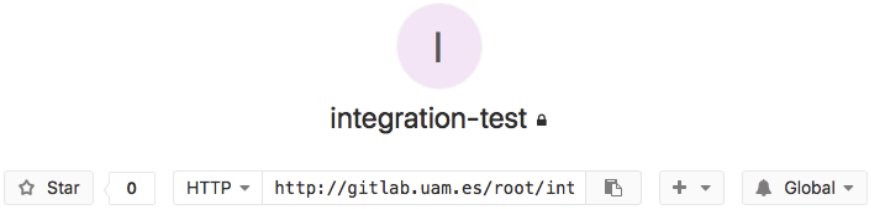

Ilustración 15. Alta de repositorio en Gitlab.

A continuación, comprobamos desde el *dashboard* que los *runners* están activos

Type	Runner token	Description	Version	IP Address	Projects	Jobs	Tags	Last contact
shared	d9556e8c	gitlab-ci-runner-1	10.4.0	172.16.0.8	n/a	0		less than a minute ago
shared	84f7b9d5	gitlab-ci-runner-0	10.4.0	172.16.0.8	n/a	0		less than a minute ago

Ilustración 16. Vista de los runners desde el dashboard.

Como vemos la IP corresponde al nodo HAProxy en los dos runners. Esto se debe a que las peticiones de los dos nodos de Gitlab se encuentran balanceados.

En nuestro repositorio, es necesario la creación de los ficheros de configuración del CI (*.gitlab-ci.yml*) que se encargan de declarar los pasos del pipeline de la aplicación.

```
# Select image from https://hub.docker.com/_/php/
image: php:5.6

# Select what we should cache
cache:
  paths:
  - vendor/

before_script:
# Install git, the php image doesn't have installed
- apt-get update -yqq
- apt-get install git -yqq

# Install mysql driver
- docker-php-ext-install pdo_mysql

# Install composer
- curl -sS https://getcomposer.org/installer | php

# Install all project dependencies
- php composer.phar install

services:
- mysql

variables:
  # Configure mysql service (https://hub.docker.com/_/mysql/)
```

```
  MYSQL_DATABASE: hello_world_test
  MYSQL_ROOT_PASSWORD: mysql

# We test PHP5.6 (the default) with MySQL
test:mysql:
  script:
  - vendor/bin/phpunit --configuration phpunit_mysql.xml --coverage-text

# We test PHP7 with MySQL, but we allow it to fail
test:php7:mysql:
  image: php:7
  script:
  - vendor/bin/phpunit --configuration phpunit_mysql.xml --coverage-text
  allow_failure: true
```

Tras escribir un programa de ejemplo en PHP, ya podemos dar de alta el clúster de Kubernetes en el repositorio para que lance las pipelines.

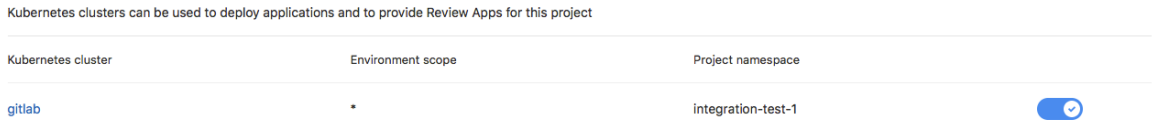

Ilustración 17. Alta de Kubernetes en el repositorio de GitLab

Cada vez que se lance un *commit* dentro del repositorio, el pipeline se lanza de manera automática. Para comprobarlo, realizamos un simple cambio en README del proyecto. Como vemos en la siguiente figura, los Jobs se ejecutan correctamente:

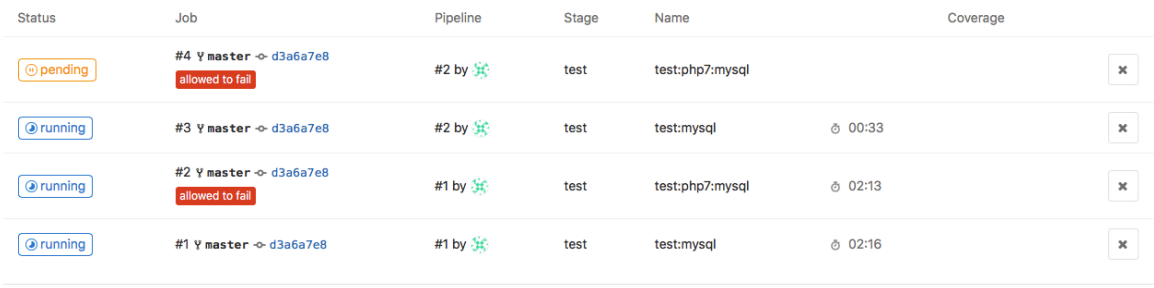

Ilustración 18. Ejecución de los runners tras un commits.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

El proyecto tenía como objetivo, la implantación de una plataforma en alta disponibilidad, basada en una aplicación de control de versiones.

Con respecto ha ido evolucionando el proyecto, nos hemos dado cuenta que la plataforma es de ámbito general, esto quiere decir, que en la parte de *frontend* (capa de aplicación) podría utilizarse cualquier tecnología. La infraestructura de la solución final está limitada a una prueba de concepto. En un entorno productivo, sería más conveniente el uso de tecnologías más estables KVM, LXD o infraestructura en la nube. La persistencia de datos en este tipo de sistemas distribuidos es crítica, por ello, habría que darle una especial importancia a la distribución y replicación de discos.

El proceso de automatización, ha requerido un gran esfuerzo inicial de comprensión y desarrollo de los *scrips* para los despliegues. Según iba avanzando el proyecto, han sido una herramienta clave para ahorrar tiempo en duplicación de código. Creemos que el código desarrollado es perfectamente reutilizable en otros proyectos de circunstancias similares.

La aplicación de GitLab ha demostrado ir muy por delante del resto de tecnologías de control de versiones integrándose con los requisitos del panorama actual en las TIC.

6.2 Trabajo futuro

Los aspectos a mejorar son:

- Implementación del clúster de base de datos.
- Proponer sistemas de virtualización más estables.
- Realizar pruebas de estrés.
- Mejorar la securización de red y de usuario.

7 Referencias

[1] K. Jackson, C. Bunch y E. Sigler, OpenStack Cloud Computing Cookbook, Pack Pub, 2015.

[2] HAProxy, «haproxy.org,» 20 06 2018. [En línea]. Available: http://www.haproxy.org. [Último acceso: 10 06 2018].

[3] J. Humble y D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley Signature Series (Fowler), 2010.

[4] J. Lewis y F. Martin, «Microservices,» *https://martinfowler.com/articles/microservices.html,* 2014.

[5] «Ansible Documentation,» [En línea]. Available: https://docs.ansible.com/.

[6] «Docker,» [En línea]. Available: https://www.docker.com/.

[7] «Aqua,» Marzo 2018. [En línea]. Available: https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016.

[8] «GitLab,» [En línea]. Available: https://about.gitlab.com/high-availability/.

[9] «Redis,» [En línea]. Available: https://redis.io/documentation.

[10] «Kubernetes,» [En línea]. Available: https://kubernetes.io/docs/setup/independent/high-availability/#reliable-nodes.

Glosario

API	Application Programming Interface
REST	Representational State Transfer
CD/CI	Continuous delivery/ Continuous integration
DTOs	Data transfer objects
LXD	Linux containers hypervisor
LXC	Linux containers
HTTP	Hypertext Transfer Protocol
URI	Uniform Resource Identifier
VIP	Virtual IP
SSH	Secure SHell
VLAN	Virtual LAN

Anexos

Anexo A Vagrantfile completo para desplegar la infraestructura

Este anexo muestra la configuración del fichero Vagrantfile.j2. Este fichero es un *template* de Ansible con la intención de parametrizar las variables correspondientes a la infraestructura.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
domain   = 'GitlabHA'

nodes =
[
{% for server in servers %}
  {   :hostname   =>   '{{server.name}}',   :ip   =>   '{{server.ip}}',   :box   =>
'{{server.os}}', :memory => {{server.memory}}},
{% endfor %}
]

Vagrant.configure("2") do |config|
  nodes.each do |node|
    config.vm.define node[:hostname] do |nodeconfig|
      nodeconfig.vm.box = node[:box]
      nodeconfig.vm.box_url =  "https://vagrantcloud.com/" + node[:box]
      nodeconfig.vm.hostname = node[:hostname]
      nodeconfig.vm.network   :private_network,   ip:   node[:ip],   :name   =>
'vboxnet3', :adapter => 2, :auto_config => true


      nodeconfig.vm.provider :virtualbox do |vb|
        vb.customize [
          "modifyvm", :id,
          "--natnet1", "192.168.222.0/24",
          "--memory", node[:memory],
        ]
      end
    end
  end

  config.vm.provision "ansible" do |ansible|
    ansible.verbose = "v"
    ansible.playbook = "playbook-init.yml"
    ansible.extra_vars = {
      ansible_python_interpreter: "/usr/bin/python3",
    }
    ansible.groups = {
      {% for groups in inventory %}
      "{{groups.group}}" => [{% for item in groups.hosts %}"{{item}}",{% endfor
%}],
      {% endfor %}
    }
  end
end
```

Anexo B Principales tareas de instalación y configuración.

```
- hosts: all
  become: yes
  become_method: sudo
  gather_facts: yes

- name: INSTALL HAPROXY NODE
  hosts: haproxy
  become: yes
  tags: [haproxy]
  roles:
    -  role: ansible-role-haproxy

- name: INSTALL GLUSTERFS NODE
  hosts: glusterfs
  become: yes
  tags: [gluster]
  roles:
    -  role: ansible-role-glusterfs
       gluster_role: server

- name: INSTALL GITLAB NODE
  hosts: gitlab-master
  become: yes
  tags: [gitlab_master,gitlab,gluster]
  roles:
    -  role: ansible-role-glusterfs
       gluster_role: client
    -  role: ansible-role-gitlab
       gitlab_role: master

- name: INSTALL GITLAB NODE
  hosts: gitlab-slave
  become: yes
  tags: [gitlab_slave,gitlab,gluster]
  roles:
    -  role: ansible-role-glusterfs
       gluster_role: client
    -  role: ansible-role-gitlab
       gitlab_role: slave

- name: INSTALL DOCKER
  hosts: kubernetes
  gather_facts: true
  become: yes
  tags: [docker,kubernetes]
  roles:
    -  role: ansible-role-docker
       tags: [docker]

- name: INSTALL KUBERNETES MASTER
  hosts: kubernetes-master
  become: yes
  gather_facts: true
  tags: [kubernetes]
  roles:
    -  role: ansible-role-kubernetes
       kubernetes_role: master

- name: INSTALL KUBERNETES NODE
  hosts: kubernetes-node
  become: yes
  tags: [kubernetes]
  roles:
    -  role: ansible-role-kubernetes
       kubernetes_role: node
```

Anexo C Fichero de configuración de los runner en kubernetes

Generación de namespace

```
apiVersion: v1
kind: Namespace
metadata:
  name: {{ gitlab_namespace }}
```

Variables globales de la imagen

```
apiVersion: v1
data:
  REGISTER_NON_INTERACTIVE: "true"
  REGISTER_LOCKED: "false"
  CI_SERVER_URL: "{{gitlab_ci_server_url}}"
  METRICS_SERVER: "0.0.0.0:9100"
  RUNNER_REQUEST_CONCURRENCY: "4"
  RUNNER_EXECUTOR: "kubernetes"
  KUBERNETES_NAMESPACE: {{ gitlab_namespace }}
  KUBERNETES_PRIVILEGED: "true"
  KUBERNETES_CPU_LIMIT: "1"
  KUBERNETES_MEMORY_LIMIT: "512Mi"
  KUBERNETES_SERVICE_CPU_LIMIT: "1"
  KUBERNETES_SERVICE_MEMORY_LIMIT: "512Mi"
  KUBERNETES_HELPER_CPU_LIMIT: "500m"
  KUBERNETES_HELPER_MEMORY_LIMIT: "100Mi"
  KUBERNETES_PULL_POLICY: "if-not-present"
  KUBERNETES_TERMINATIONGRACEPERIODSECONDS: "10"
  KUBERNETES_POLL_INTERVAL: "5"
  KUBERNETES_POLL_TIMEOUT: "360"
kind: ConfigMap
metadata:
  labels:
    app: gitlab-ci-runner
  name: gitlab-ci-runner-cm
  namespace: {{ gitlab_namespace }}
```

Generación del *token* de acceso a Gitlab

```
apiVersion: v1
kind: Secret
metadata:
  name: gitlab-ci-token
  namespace: {{ gitlab_namespace }}
  labels:
    app: gitlab-ci-runner
data:
  GITLAB_CI_TOKEN: {{ gitlab_encrypt_token.stdout }}
```

Declaración de la aplicación

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: gitlab-ci-runner
  namespace: {{ gitlab_namespace }}
  labels:
    app: gitlab-ci-runner
spec:
  updateStrategy:
    type: RollingUpdate
  replicas: 2
  serviceName: gitlab-ci-runner
  template:
    metadata:
      labels:
```

```
        app: gitlab-ci-runner
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - topologyKey: "kubernetes.io/hostname"
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - gitlab-ci-runner
     volumes:
        - configMap:
            name: gitlab-runner
          name: config
        - hostPath:
            path: /usr/share/ca-certificates/mozilla
          name: cacerts
      serviceAccountName: gitlab-ci
      securityContext:
        runAsNonRoot: true
        runAsUser: 999
        supplementalGroups: [999]
      containers:
      - image: gitlab/gitlab-runner:v10.4.0
        name: gitlab-ci-runner
        command:
        - /scripts/run.sh
        envFrom:
        - configMapRef:
            name: gitlab-ci-runner-cm
        - secretRef:
            name: gitlab-ci-token
        env:
        - name: RUNNER_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        ports:
        - containerPort: 9100
          name: http-metrics
          protocol: TCP
        volumeMounts:
        - mountPath: /etc/gitlab-runner
          name: config
        - mountPath: /etc/ssl/certs
          name: cacerts
          readOnly: true
      restartPolicy: Always
```